MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo No. 684 December, 1982

A Subdivision Algorithm in Configuration Space for Findpath with Rotation

Rodney A. Brooks and Tomás Lozano-Pérez

Abstract. A hierarchical representation for configuration space is presented, along with an algorithm for searching that space for collision-free paths. The details of the algorithm are presented for polygonal obstacles and a moving object with two translational and one rotational degrees of freedom.

Acknowledgements. This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's Artificial Intelligence research is provided in part by the Office of Naval Research under Office of Naval Rescarch contract N00014-81-K-0494 and in part by the Advanced Research Projects Agency under Office of Naval Research contracts N00014-80-C-0505 and N00014-82-K-0334.

1. Introduction

In this paper we present an algorithm for finding collision-free paths for a rigid polygonal object moving through space that is cluttered with obstacle polygons. The paths can include rotations of the object. The algorithm will find a path from a given initial position and orientation to a goal position and orientation if such a path exists, subject only to a user-specified resolution limit on displacements.

The problem addressed here is an instance of the problem known as the *findpath* or *mover's problem* in robotics. The problem arises when planning the motion of a robot manipulator or mobile robot in an environment with known obstacles. For related approaches to the findpath problem, see Brooks [1982], Lozano-Pérez [1981, 1983], Lozano-Pérez and Wesley [1979], Moravec [1980], Schwartz and Sharir [1981, 1982], and Udupa [1977]. A survey of the different approaches to findpath and a discussion of its role in robot task planning can be found in Lozano-Pérez [1983].

The algorithm described here is based on the configuration space approach described by Lozano-Pérez [1981, 1983]. The *configuration* of a rigid object is a set of independent parameters that characterize the position of every point in the object. We associate a local coordinate frame with a rigid object, such as a planar polygon. The configuration of the polygon can be specified by the x, y position of the origin of the local coordinate frame, known as the *reference point* and a θ value indicating the rotation of the local frame relative to the global frame. The space of all possible configurations of an object is its *configuration space*. A point in the configuration space, a *configuration point*, represents a particular position of the object's reference point and an orientation of the object's axes.

The configuration space for planar polygons is three-dimensional while that of solid polyhedra is six-dimensional: three translational and three rotational dimensions. Due to the presence of the immovable obstacles some regions of the configuration space are not reachable; these regions are the *configuration obstacles*. Hence, in the configuration space, the moving object is shrunk to a configuration point while the immovable obstacles are

expanded to fill all space where the presence of the configuration point would imply a collision of the object with obstacles. The findpath problem of finding a path for the object through the original space while avoiding obstacles is thus transformed to finding a path for the configuration point through the configuration space while avoiding the configuration obstacles.

The fundamental structure of the algorithm is extremely simple. Configuration space is first divided into rectangloids with sides normal to the axes of the space. Each rectangloid is labeled as (1) *empty* if the interior of the rectangloid nowhere intersects a configuration obstacle, (2) *full* if the interior of the rectangloid everywhere intersects the configuration obstacles or (3) *mixed* if there are interior points both inside and outside of configuration obstacles. A free path is found by first finding a connected set of *empty* rectangloid cells that include the initial and goal configurations and constructing a piecewise linear path through those *empty* cells. If such an *empty* cell path cannot be found in the initial subdivision of configuration space then a path that includes *mixed* cells is found. *Mixed* cells on the path are subdivided, by cutting them with a single plane normal to a coordinate axis, and each resulting cell is appropriately labeled as *empty*, *full*, or *mixed*. Another search for an *empty*–cell path is initiated, and so on iteratively until success is achieved. If at any time no path can be found through non–*full* cells of greater than some preset minimum size, then the problem as posed is insoluble (i.e., no collision free path exists given the resolution limit).

The conceptual and practical problems that must be solved to implement this algorithm are as follows:

(a) What is an efficient algorithm for labeling the newly cut cells as *empty*, *full* or *mixed*?

(b) How should configuration space be initially divided into rectangloids?

(c) Where should a *mixed* cell be cut when it is subdivided?

(d) What additional information should be kept with *mixed* cells describing the configuration space obstacles they intersect?

(e) How should the iterative search be controlled?

Section 2 answers questions (a) through (d) for the case of planar polygons with no rotations allowed. It develops a grid based representation for configuration space, and develops some key operations on that representation.

Section 3 shows how the representation can be simply extended to handle rotations of the moving object – the representation of configuration space is extended into three dimensions.

Section 4 discusses question (e) and describes a procedure that produces a reasonably efficient search.

Section 5 shows results and discusses some of the limitations of the algorithm.

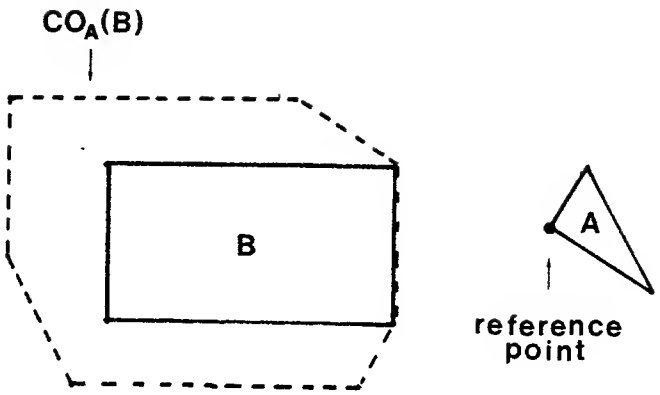


Figure 1. A configuration obstacle, $CO_A(B)$, for convex polygons A and B

2. Representation of Configuration Space Obstacles

For a convex polygonal moving object A and a convex polygonal obstacle B we write the configurations forbidden for A as $CO_A(B)$. Figure 1 shows such a configuration obstacle for polygons A and B where their relative orientation is fixed. Lozano–Pérez [1983] showed that $CO_A(B)$, for fixed relative orientations of convex A and B, is also a convex polygon. A convex polygon can be expressed as a conjunction of linear inequalities.

When changes in the orientation of the moving object are allowed, the resultant $CO_A^\theta(B)$ are neither convex nor bounded by half-spaces. Their surfaces are curved, although they do have the property that when cut by a plane parallel to the rotation dimension they have polygonal cross sections (since such a cross section corresponds to a single fixed orientation).

Lozano–Pérez [1983] showed that for two dimensional A and B the surfaces of the configuration obstacle could be expressed as inequalities each valid over an infinite rectangloid R in (x, y, θ) space, where

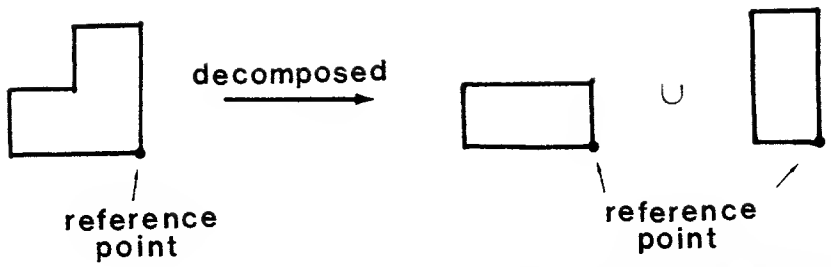


Figure 2. Non-convex moving objects can be decomposed into a union of convex objects sharing a common reference point and reference orientation.

$$R = [-\infty, \infty] \times [-\infty, \infty] \times [\theta_1, \theta_2]$$

for some fixed θ_1 and θ_2. The forms of these surfaces are given in section 3.3.

Hence, either with or without rotation for a single convex obstacle and a convex moving object we can cut up configuration space into rectangloids R_i. The subset of the configuration space obstacle that intersects R_i can be defined by those points in R_i such that $\bigwedge_i e_i$ is satisfied, where each e_i is an inequality.

In the more general case the moving object A is represented as a union of (possibly overlapping) convex polygons. Similarly, multiple obstacles are each represented as unions of convex polygons. Let the convex moving polygons be $A_1, A_2, \ldots, A_n$. Define a single reference point P and reference orientation for all the A_i. See Figure 2 for an example. Let the convex obstacle polygons be $B_1, B_2, \ldots, B_m$.

The path for object A can be found through the obstacle littered space by finding a path for the configuration point through configuration space filled with the union of the $CO_{A_i}(B_j)$'s or $CO^{\theta}_{A_i}(B_j)$'s. Hence, for arbitrary obstacle and moving object polygons, we first decompose them into the unions of convex polygons; then, for all pairs of obstacle and object polygons we compute the configuration space obstacles. These new obstacles are all embedded in the same configuration space, and a collision–free path is found through it for

the configuration point of the object to be moved.

In the remainder of this section we develop the representation of configuration space for 2 dimensions with fixed orientation. Section 3 generalizes to 2 dimensions with rotations.

2.1 Representation of Configuration Space.

Individual configuration obstacles can be represented as a conjunction of inequalities

$$\bigwedge_i e_i.$$

We refer to these inequalities as *constraints*. Each one has the form $f(x) \leq 0$, where x is the configuration point for the moving object. A point x is *inside* such a constraint if $f(x) < 0$, *outside* if $f(x) > 0$ and *on* the constraint if $f(x) = 0$. A point is contained in a configuration obstacle if it is not *outside* any of its defining constraints. A point x is on the surface of an obstacle if it is *on* some constraint.

A configuration obstacle can be bounded by a finite rectangloid box in configuration space with edges parallel to the coordinate axes. The obstacle embedded in configuration space can then be represented as the box along with a conjunction of constraints. If two obstacles are embedded in the same space and the boundary boxes intersect then the two boxes can be decomposed into a union of boxes. Each resulting box has associated with it a disjunction of the conjunctions representing the original two obstacles. Of course, the sub-boxes that only overlapped a single configuration space obstacle have just the original conjunction.

More formally, we represent configuration obstacles as follows. Configuration space is subdivided into rectangloid *cells* in order to represent the embedding of many configuration obstacles. Each cell is a closed rectangloid with edges parallel to the coordinate axes. The union of all cells is the whole space and no point is in the interior of two cells. Associated with each cell is a *sentence* of constraints of the form

$$\bigvee_{i=1}^{k} \bigwedge_{j=1}^{l_i} e_{ij}.$$

Each conjunction of constraints in a sentence will be referred to as a *clause* and each clause is made up of *terms*, i.e., each constraint will be referred to as a term.

A point is said to be *inside* sentence S if for some clause of the disjunction it is not *outside* any of the constraints. A point is said to be *outside* sentence S if it is *outside* some constraint in every clause. A cell C with associated sentence S is labeled as

(1) *empty* if no point of C is *inside* S,

(2) *full* if no point of C is *outside* S,

(3) *mixed* otherwise.

The labeling is done as a by–product of trying to simplify sentence S; see below.

2.2 Simplification of Constraint Sentences.

Both while constructing the original representation of configuration space and when subdividing cells during search it is necessary to associate a constraint sentence S with a cell C. It may be the case that the cell is completely *outside* or completely *inside* some constraints in S. The procedure below both simplifies S by removing such constraints and, as a by–product, labels the cell *empty*, *full*, or *mixed*.

```
procedure SimplifyAndLabel (C, S);
begin
  foreach CLAUSE in S do
    begin
      foreach TERM in CLAUSE do
        begin
          case CellCompare (C, TERM) of
               Inside: remove TERM from CLAUSE;
               Outside: begin
                          remove CLAUSE from S;
                          goto next CLAUSE;
                          end;
               Cut: ;
            endcase;
          end;
      if CLAUSE is nil
         then begin
                label C with full
                set S to nil;
                exit from SimplifyAndLabel;
                end;
      end;
  if S is nil
     then label C with empty
     else label C with mixed;
  end;
```

Figure 3 illustrates the behavior of the algorithm.

A sub–procedure CellCompare is used to compare a cell C with a single constraint, e say. It returns one of:

(1) Outside if no point of C is *inside* e,

(2) Inside if no point of C is *outside* e,

(3) Cut otherwise.

The formulation of CellCompare depends on the form of constraints to be considered. Note also that in general the surface of a constraint (i.e., those points *on* the constraint) is of lower dimension than the configuration space in which it is embedded. Thus a constraint

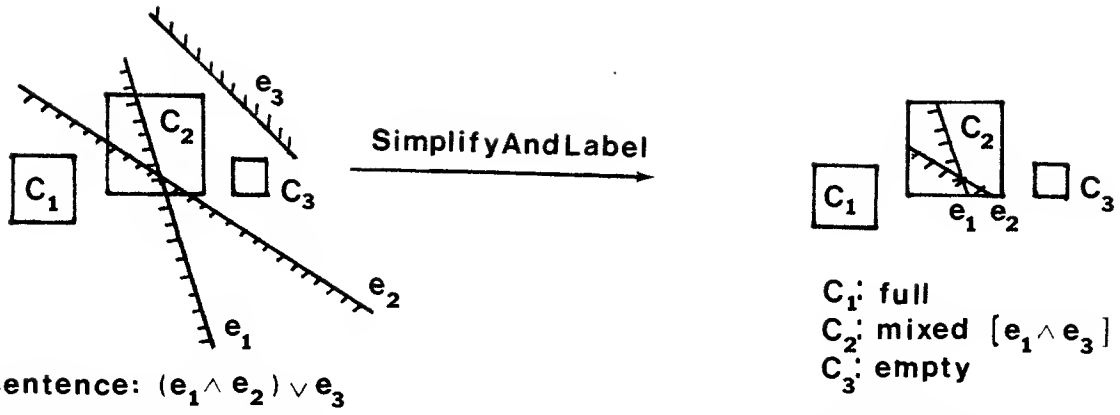


Figure 3. The SimplifyAndLabel procedure labels cells and simplifies constraint sentences of cells.

fills only a subset of measure zero of a cell.

2.3 Comparing a Cell to a Single Constraint.

When no rotations are allowed all the constraints describing a configuration obstacle are linear. To decide whether a cell is cut by a linear constraint, or if it lies on at most one side of the constraint, it suffices to test whether all the vertices of the cell are on the same side of the constraint. This test can be generalized to testing the vertices of convex polygonal cells.

2.4 Bounding Configuration Obstacles.

Let (b_{ix}, b_{iy}) be vertices of B, and (a_{jx}, a_{jy}) are vertices of A relative to reference point P. Lozano-Pérez [1983] showed that $CO_A(B)$, in two dimensions without rotations and for convex A and B, is the convex hull of all the points of the form

$$(b_{ix} - a_{jx}, b_{iy} - a_{jy})$$

Hence, $CO_A(B)$ can be bounded by the rectangle

$$[x_l, x_h] \times [y_l, y_h]$$

where

$$\begin{aligned} x_l &= \min_{i,j}(b_{ix} - a_{jx}) \\ x_h &= \max_{i,j}(b_{ix} - a_{jx}) \\ y_l &= \min_{i,j}(b_{iy} - a_{jy}) \\ y_h &= \max_{i,j}(b_{iy} - a_{jy}). \end{aligned}$$

2.5 Constructing an Initial Representation of Configuration Space.

We are now in a position to construct an initial representation of configuration space. There are two operations. (1) Cut space into rectangloid cells. (2) Build a *connectivity graph* between the cells. By combining the second operation with the first it is possible to reduce its complexity to linear in the final number of cells.

Figure 4 shows a configuration space cut into rectangular cells. Figure 5 shows its corresponding connectivity graph.

A cell is represented by a rectangle, a constraint sentence in constraints that cut it, a label from among *full*, *empty*, and *mixed*, and pointers to neighboring cells. The neighbors are grouped according to their direction, e.g. $+x$, $-x$, $+y$, and $-y$ for two dimensional configuration space. During construction of the initial representation of configuration space, the elements within these groupings may be ordered. Temporarily, a cell can also contain a list of configuration obstacle bounding rectangles; see below.

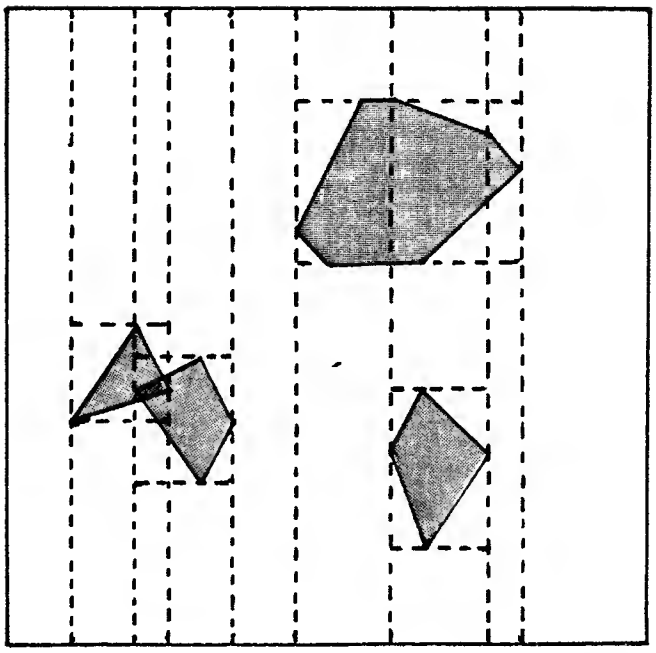


Figure 4. Configuration space with obstacles cut into rectangular cells.

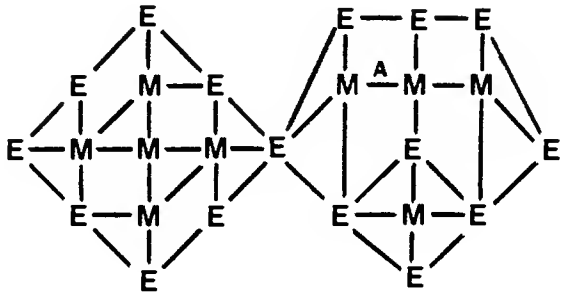


Figure 5. Connectivity graph for cells in Figure 4. The link labeled A is not actually included in the graph as it is impossible to traverse it.

The cutting algorithm proceeds as follows. An order is chosen for the coordinates of the configuration space. For illustrative purposes assume it is x followed by y. For the first of these coordinates (x) the bounding rectangles for each $CO_{A_i}(B_j)$ are sorted on their lower bound in that coordinate. Rectangles with the same lower bound are sorted on their upper bounds.

The list of values of the first coordinate for the upper and lower bounds of all bounding rectangles is sorted and used to divide the configuration space into large cells (strips) with boundaries at each of these values. The neighborhood relations formed due to cutting are trivially computed, and newly cut strips are linked to their two neighbors, in the $-x$ and $+x$ directions. As the list of bounding rectangles is scanned, any that overlap a strip are cut at the strip boundaries. The constraint sentence of the corresponding configuration obstacle is associated with the new bounding rectangle and, then, simplified by the procedure SimplifyAndLabel. Any *empty* bounding rectangle can be removed from further consideration. The new bounding rectangles with simplified sentences are associated with the newly formed strips.

The second stage applies almost the same procedure recursively to each strip, cutting the strips along the second coordinate (y in this case). The new bounding rectangles associated with the strips in the first pass provide the cutting values. The strips are considered in ascending order of their lower x bounds. Cutting the first (least x) strip proceeds much as before except that every new cell has the second strip as its $+x$ neighbor. The $-x$ neighbor relation in the second strip is replaced by pointers to the new cells derived from the first strip, ordered by increasing value of y. The $-y$ and $+y$ neighbor relations between newly cut cells are trivially computed.

Subsequent strips are similarly divided, but now, besides scanning the bounding rectangles in order of increasing value of their lower y bounds, the neighbors produced by cutting the previous strip are also scanned. The ordered $-x$ neighbor relations of the subsequent strip are used for this scan. This process is illustrated in figure 6. It shows some detail of the construction of figures 4 and 5. The third strip is cut in the y direction

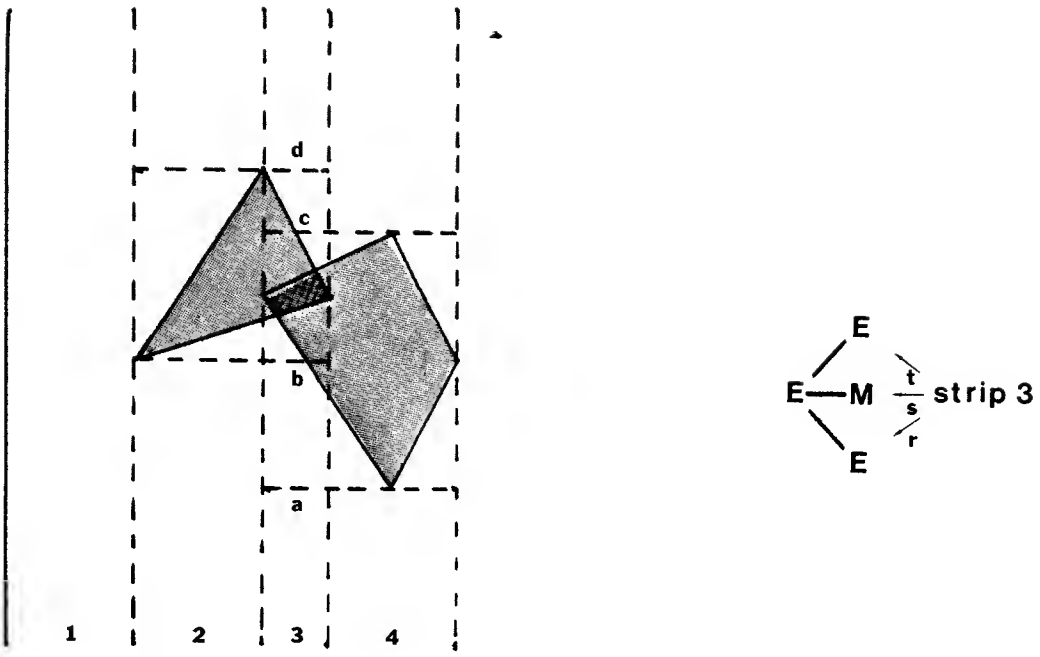


Figure 6. As strips are cut in the second direction an ordered list of pointers to the cells produced in the previous strip is followed producing the neighbor relations in linear time.

following the cutting of the first two strips. The partially constructed connectivity graph includes directed arcs from strip 3 to the 3 cells of strp 2. An ordered list of cuts, a, b, c and d, is scanned in parallel with the ordered list of arcs r, s and t. The scan makes use of two pointers one into each list; each pointer is incremented until it passes the other pointer, and then that pointer is incremented. Whenever the pointer into the cut list is incremented, a cut is made and the new cell uses the other pointer to determine its connectivity graph neighbor in the $-x$ direction. At the same time a new ordered list of pointers from strip 4 into the new cells is constructed for use when cutting that strip. This process is iterated for all the strips in order of increasing x lower bound. In general this procedure can be applied recursively to as many dimensions as the space has.

During the final cutting operation when the last coordinate (y in this case) is being used, the bounding rectangles are handled differently. Instead of associating the new bounding

rectangle directly with the newly cut cells of configuration space, their constraint sentences are associated with the cell. The constraint sentences of all the bounding rectangles that overlap a cell are combined disjunctively and the result becomes the constraint sentence of the configuration space cell. The SimplifyAndLabel procedure is applied to the new cell and this disjunctive sentence. Any cell so labeled *full* need not be linked into the connectivity graph. The result of the final operation is shown for the example in Figure 5.

A refinement of the above procedure is prompted by the following observation. Two neighboring cells C_1 and C_2 intersect in a rectangloid cell C one dimension lower than the original cells. Both constraint sentences S_1, from C_1, and S_2, from C_2, apply to the intersection cell. If either SimplifyAndLabel(C, S_1) or SimplifyAndLabel(C, S_2) label C as *full* then the configuration point cannot move directly from cell C_1 to C_2. Hence, although spatially neighbors, none of their e*mpty* interior points are connected by a path for the moving object's configuration point. Thus their neighbor relation can be omitted from the connectivity graph. For this reason, link A of Figure 5 can be deleted. This refinement significantly cuts down the number of links in realistic connectivity graphs, greatly increasing efficiency.

In two–dimensional configuration space there is no inherent advantage to choosing one of x or y as the initial direction in which to cut space. In higher dimensions this will become an issue.

A major drawback to the described cutting operation is that a number of small obstacles localized in one part of space can have significant global effects on the representation of space elsewhere, e.g., the effect of x bounds in Figure 4. This can become a significant problem in three–dimensional configuration space and more refinements to the above procedure are introduced in section 3.5 to deal with this.

2.6 Deciding Where to Cut a Cell.

During search, the representations of *mixed* cells that lie on the candidate path are refined.

A cell C with sentence S is cut into two cells, C_1 and C_2, by splitting along one of its coordinates. Then, each sub–cell is labeled by calling SimplifyAndLabel(C_i, S) for $i = 1, 2$.

The cutting operation can help the search converge on a path through *empty* cells but only if it makes it easier to deduce if at least one of the C_i's is *full* or *empty*. This will be the case if the number of constraints appearing in at least one sentence S_i is reduced from the number that appeared in S. Such simplification can be achieved if one of the new cells lies wholly *inside* or *outside* a constraint. We refer to it as the simpler sub–cell.

Two heuristic principles have also been used in the implemented algorithms described here. First, it is desirable that the new cell with simpler constraint sentence should have the maximal volume possible. This offers the possibility of finding that a large volume is either *empty* or *full*. In one case it makes the search for a free path easier and in the other eliminates a large volume from further consideration. The second heuristic principle is that large amounts of computation in choosing the place to cut a cell, would be better spent in cutting the cell into more, less well chosen, pieces, as more cuts result in greater likelihood of actually finding *empty* or *full* sub–cells.

Thus there are three problems; (1) find cuts that lead to simpler S_1 and S_2; (2) choose the one that gives the simpler cell a large volume; (3) compute (1) and (2) quickly. Here we show how to choose a good cut according to these criteria. The method is stated in more generality than necessary so that it can easily be extended to the higher dimensional cases later in the paper.

The basic approach is as follows. Consideration is given to cutting the cell in each direction (x and y in this case) at some number of points for each constraint in the cell's constraint sentence. A score is assigned to each such plausible cut, and the cut with minimal score is chosen as best.

For each constraint, the candidate places to cut perpendicular to the x direction, say, are those y values where the constraint intersects a cell edge parallel to the x direction.

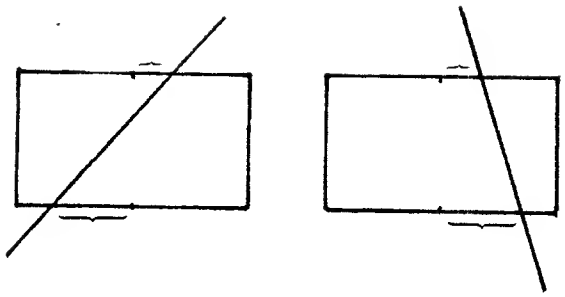


Figure 7. Given two intersection points of a constraint and boundary edges, the cell is cut at the intersection point closest to the center of the boundary edges. The two cases that are illustrated here show that this produces the simpler cell with maximal volume.

Only such points that are not at an extreme y value for the cell need be considered. There will be either zero, one or two of these points for each constraint and each chosen direction of cutting. The cuts associated with these points have the following properties.

(a) At least one of the two resulting cells is either completely *inside* or *outside* the constraint. Thus its constraint sentence will be simplified, satisfying condition (1) above.

(b) By choosing to cut at the point of intersection of the constraint and the cell edge, the volume of the simpler cell is maximized. This helps towards satisfying (2) above.

It remains now to define the scoring function. If the length of the cell in the x direction is scaled to 1.0 then the score of a cut is its distance in those units from the center in the x direction. Note that this score is a proportion of the volume of the original cell that is added or subtracted from half its volume to get the volumes of the two new cells. Thus the scale of this measure is independent of the constraint or the direction of cut.

If a given constraint and direction of cut produces two candidate cuts then the one with lowest score is the one that produces a simpler cell with maximal volume (helping in the achievement of (2)). This can be shown by considering two cases, illustrated in Figure 7.

Case 1. The two candidate cuts are on the same side of the midpoint. Then the simpler sub–cell that includes the midpoint has volume greater than half the total, so it must be the larger. The cut closest to the midpoint produces that sub–cell.

Case 2. The two candidate cuts are on opposite sides of the midpoint. The larger simpler sub–cell is the one whose cut is closest to the midpoint.

Note that, for a particular cutting direction, the simple score comparison chooses the cut for a single constraint that best satisfies condition (2) above. It does this without analysis of the constraint other than where it cuts the edges of the cell. In particular, the simpler cell was not explicitly identified, nor was its volume computed. To satisfy condition (2) globally over all constraints and cutting directions would require such an explicit volume computation, which would require a detailed analysis of the interaction of each constraint and the cell, violating condition (3).

In our implementation we choose the cut by picking the candidate with the lowest score from among all cutting directions and single constraints. This has the effect of choosing a cut that produces a simpler cell with volume closest to half the original cell volume. The chosen cell may not be the best possible choice in terms of condition (2), but it is certainly not a poor choice. If all possible simpler cells have volume less than half the original, then this algorithm will in fact choose the cut that results in the largest such cell. If there are cells bigger than half, then the biggest one might not be chosen; condition (3) is well satisfied, however.

3. Representing Rotations

The structure of the algorithms and representations for two-dimensional configuration spaces can be extended naturally to three-dimensional configuration spaces, when the moving polygon is allowed to rotate. For the most part the extensions are straightforward. There are two additional considerations in the cases where rotations are allowed, however.

(1) The cell comparison and cut selection problems are no longer linear, and so new algorithms must be developed to solve them.

(2) Rotations result in a configuration space with higher dimension than the original space in which the problem is embedded. Care must be taken to ensure the space representation does not result in unnecessary fragmentation into an inordinate numbers of cells. Such fragmentation greatly impedes search efficiency.

This section generalizes the representations and algorithms of section 2 to the case of two dimensions with rotation.

3.1 Representation of Configuration Space.

For two-dimensional obstacles with a rotating moving object, the configuration space is three-dimensional; two dimensions inherited from the original space and an orthogonal rotation dimension. The representation of configuration space in this case is a simple generalization from the two-dimensional configuration space, i.e., a union of closed rectangloid cells that have no common interior points. The sides of the rectangloids are orthogonal to the axes of the configuration space. Each cell has associated with it a label from among *full*, *empty*, and *mixed*. Each *mixed* cell also has an associated constraint sentence that is a disjunction of conjunctions of constraints. Points within the cell that satisfy the constraint sentence are forbidden points in configuration space; others represent locations and orientations of the moving object where it does not collide with the obstacles. The cells are linked in a connectivity graph, but now cells can have neighbors in either direction along

the two linear dimensions, and in each direction along the rotational dimension. The linear and rotational dimensions are treated completely uniformly within the representational scheme.

Again each constraint is of the form $f(x) \leq 0$. A point x is *inside* such a constraint if $f(x) < 0$, *outside* if $f(x) > 0$ and *on* the constraint if $f(x) = 0$. Whereas before f was a linear function, it now has trigonometric terms. Lozano-Pérez [1983] derived the two forms that these constraints can take for planar polygons:

$$f(x, y, \theta) = x\cos(\theta + \lambda_i) + y\sin(\theta + \lambda_i) - \|b_j\|\cos(\theta + \lambda_i - \gamma_j) - \|a_i\|\cos(\lambda_i - \eta_i) \quad \text{(type } A\text{)}$$

and

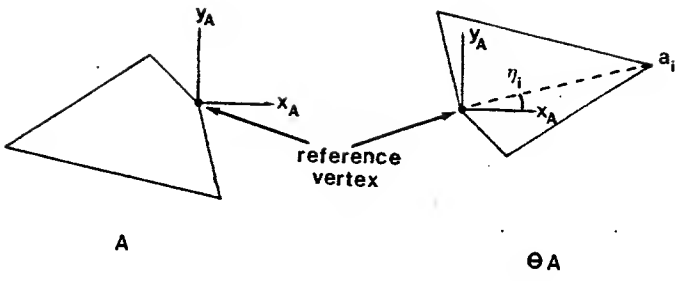
$$f(x, y, \theta) = x\cos\phi_j + y\sin\phi_j - \|b_j\|\cos(\phi_j - \gamma_j) - \|a_i\|\cos(\theta + \eta_i - \phi_j) \quad \text{(type } B\text{)}.$$

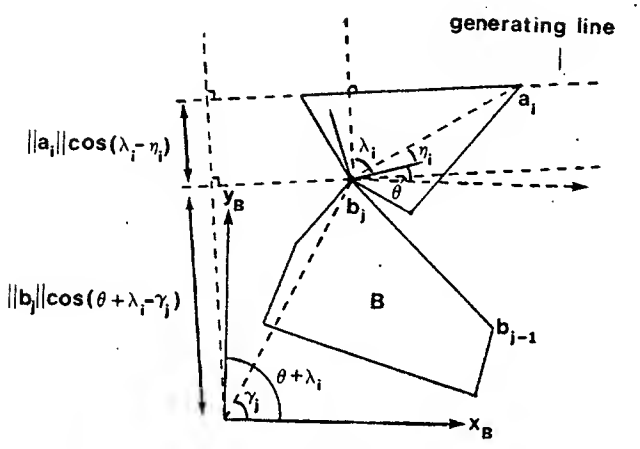
Here the a_i's are vertices of the "negated" moving polygon ($\ominus A$ in Lozano-Pérez [1981, 1983]), in its local coordinate system. η_i is the angle the line from the origin of that coordinate system to the point a_i makes with the coordinate system's x axis, and λ_i is the angle made by the normal to the segment from a_i to a_{i+1}. Similarly the b_j's are the vertices of a convex obstacle polygon, γ_j the orientation of the line from the origin to b_j, and ϕ_j the orientation of the normal to the segment from b_j to b_{j+1}. The parameter θ, a parameter of the configuration space, measures the angle between the x-axes of the object and obstacle coordinate systems.

Type A constraints can be thought of as being generated by a face (edge) of the moving object A coming into contact with a vertex of an obstacle B, and a type B constraint as a vertex of A coming into contact with a face (edge) of B.

Each constraint is valid only over a fixed range of θ. For type A surfaces the range is given by

$$\theta \in [\phi_{j-1} - \lambda_i, \phi_j - \lambda_i]$$





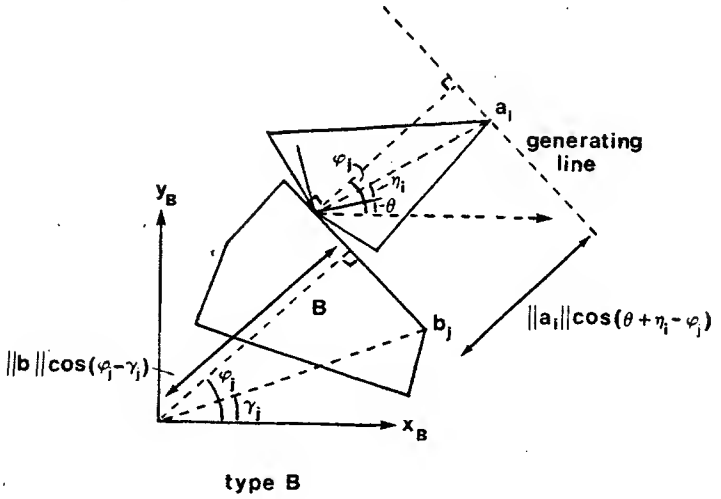


Figure 8. The two types of surfaces can be defined by bring the reference point of the negative of moving object A into contact with a vertex and an edge of fixed obstacle B. Both are defined over a range of orientations θ.

and for type B surfaces by

$$\theta \in [\phi_j - \lambda_i, \phi_j - \lambda_{i-1}].$$

A constraint is only included in the constraint sentence of a cell whose θ range lies within its range of validity.

3.2 Simplification of Constraint Sentences.

The simplification algorithm needs no generalization. It does make use of a much more complex CellCompare procedure described below, however.

3.3 Comparing a Cell to a Single Constraint.

The idea of treating the spatial and rotational components of configuration space uniformly breaks down when comparing a cell to a constraint. This is because the constraint surfaces are, in some sense, "well behaved" with respect to x and y, but "poorly behaved" with respect to θ – the surfaces can creep into, and out of a cell along an edge in the θ direction, while the vertices at both ends are on the same side of the constraint.

For a fixed θ_0 both type A and type B constraint surfaces become a single, infinite, straight line. Type A and B constraints are, therefore, ruled surfaces, which are valid over some range $[\theta_1, \theta_2]$. Consider the projections into the x–y plane of the portions of these surfaces in the range $[\theta_1, \theta_2]$ (*slice projections* in the terminology of Lozano-Pérez [1981, 1983]). Points outside the projection must support columns in the θ direction that are either completely *inside* or completely *outside* the constraint. Furthermore, there will be two disjoint regions outside the projection that correspond to these types of columns. Figure 9 illustrates the projection of both a type A and a type B constraint.

Now consider the projection of the cell into the x–y plane. If the resulting rectangle is wholly within the *inside* region of the constraint projection then the original cell is `Inside` the constraint. If it is within the *outside* region then the cell is `Outside` the constraint. If any point of the cell projection is in the interior of the constraint surface projection,

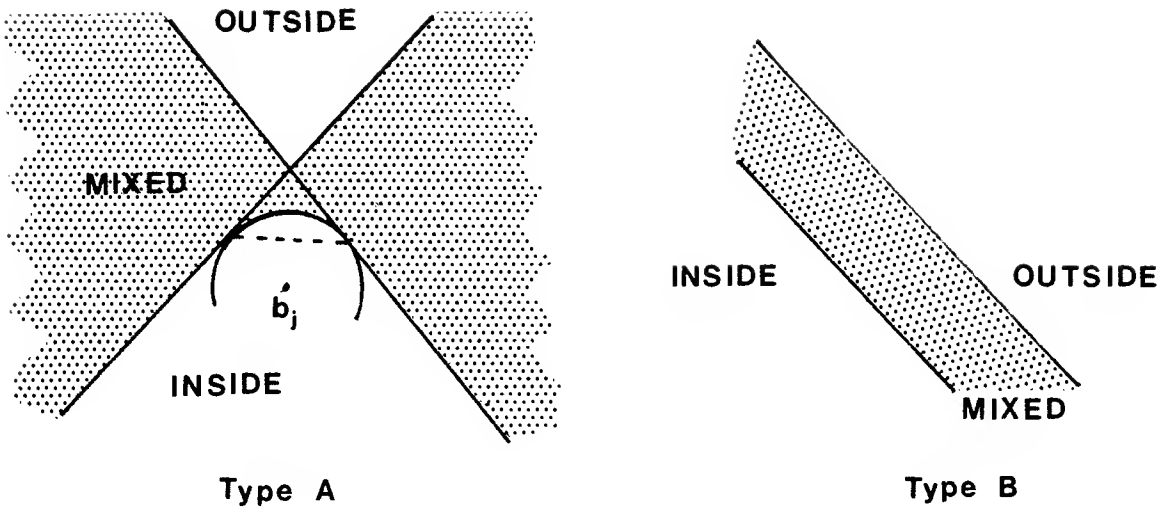


Figure 9. Slice projections into the x–y plane of Type A and Type B surfaces from configuration obstacles. These projections divide the plane into three regions: *inside*, *mixed*, and *outside*. The CellCompare procedure must determine where the projection of a cell lies relative to these regions.

however, then the cell is Cut by the constraint.

Thus the procedure CellCompare has been reduced to two more primitive operations; projection of the constraint surface into the x–y plane, and comparison of a rectangle to the two regions of the plane outside of that projection. If those regions are convex then the comparison is simple, as a convex polygon is contained in a convex set if and only if all its vertices are contained in the region.

A type B constraint projects into a strip with parallel sides. On one side is a half plane corresponding to points *inside* the constraint and on the other a half plane corresponding to points on the *outside*. Clearly these regions are both convex. The equations of the sides of the strip are:

$$x\cos\phi_j + y\sin\phi_j - \|b_j\|\cos(\phi_j - \gamma_j) - \|a_i\|\cos(\theta_a + \eta_i - \phi_j) = 0$$

and

$$x\cos\phi_j + y\sin\phi_j - \|b_j\|\cos(\phi_j - \gamma_j) - \|a_i\|\cos(\theta_b + \eta_i - \phi_j) = 0,$$

where θ_a and θ_b provide the extreme values for $\cos(\theta + \eta_i - \phi_j)$ over the range $[\theta_1, \theta_2]$. Thus they take on two of the values from the set $\{\theta_1, \theta_2, \phi_j - \eta_i\}$ (so long as the third angle lies between the first two). It is an extremely simple computation to determine whether all the vertices of the projected cell lie in one of the two half planes.

Type *A* constraints are considerably more difficult to deal with, as the orientation of the ruled line on the constraint surface rotates with changing θ. Here we carry out the analysis for the case that $\cos(\lambda_i - \eta_i) \geq 0$. The opposite case is similar with some sign changes, and a change in the roles of *inside* and *outside*. (The latter case can only occur when the reference point is outside of the convex moving object, a situation that commonly arises when a non–convex moving object is decomposed into several convex sub–objects.)

Consider Figure 9. The projection of a type *A* surface is characterized by two straight lines whose normals have orientations $\theta_1 + \lambda_i$ and $\theta_2 + \lambda_i$, and whose normal distances from the point b_j are equal to $\|a_i\|\cos(\lambda_i - \eta_i)$. A circle of that radius, centered at b_j, completes the construction. Note that both the *inside* and *outside* regions are convex. Thus to test whether a cell is *inside* or *outside* a constraint it suffices to test whether all the vertices of its projection into the x–y plane lie in one region or another.

A point is in the *outside* region if it is *outside* both of the straight lines. It is *inside* if it is *inside* both of the straight lines and not in the small area between the circle and the point of intersection of the two lines. Such points have distance from b_j greater than the radius of the circle, and are on the opposite side of the chord (in figure 9 the dashed line joining the two intersections of the circle with the straight lines) as point b_j. That two points are on opposite sides of the chord can be determined by comparing the signs of the dot product of a normal to the chord, with vectors from one end of the chord to the two points. One end of the chord has coordinates

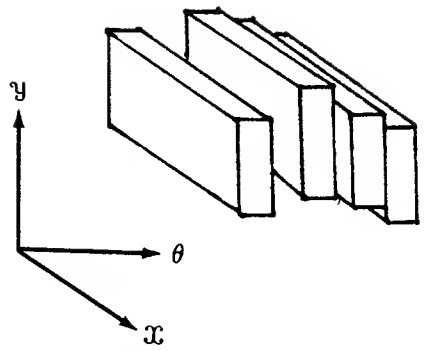


Figure 10. A "stack" of cells used to bound a three–dimensional configuration obstacle.

$$(b_{jx} + \|a_i\|\cos(\lambda_i - \eta_i)\cos(\theta_1 + \lambda_i), \quad b_{jy} + \|a_i\|\cos(\lambda_i - \eta_i)\sin(\theta_1 + \lambda_i))$$

and a normal to the chord is

$$(\sin(\theta_1 + \lambda_i) - \sin(\theta_2 + \lambda_i), \quad \cos(\theta_2 + \lambda_i) - \cos(\theta_1 + \lambda_i)).$$

3.4 Bounding Configuration Obstacles.

As θ varies, the x and y bounds on the cross section of $CO_A^\theta(B)$ normal to the θ axis varies considerably. Hence, instead of using a single bounding rectangloid, $CO_A^\theta(B)$ is represented by a "stack" of cells in the θ direction; see Figure 10. This significantly improves the approximation. The stack of cells is obtained by, first, determining a number of sub–intervals of the complete θ range and, second, determining the x and y bounds of the configuration obstacles over each sub–interval.

The sub–intervals are chosen as follows. The interval $[-\pi, \pi]$ is cut at all values of the form $2n\pi + \lambda_i - \phi_j$ that, for some integer n, lie within the interval. Any constraint that is valid for some interior point of one of the resulting sub–intervals is taken as valid over the whole closed sub–interval.

A conjunction of all valid constraints over a sub–interval forms the initial constraint sentence of a cell whose θ range corresponds to the sub–interval. Later the cells will be embedded in the configuration space, and will perhaps be intersected with other cells, whence their constraint sentences will become disjunctive terms of new constraint sentences.

For a cell whose θ interval is $[\theta_1, \theta_2]$, the maximal x value, achieved over the points that satisfy its constraint sentence, must occur at a point that has a θ component within that interval. Let the θ component of this maximal point be θ_0. By convexity, the maximal point must be a vertex of the convex polygon that is the cross section of $CO_A^\theta(B)$ at θ_0. For a fixed θ_0, Lozano–Pérez [1983] showed that this cross section is the convex hull of all points of the form

$$(b_{jx} + a_{ix}\cos\theta_0 - a_{iy}\sin\theta_0, \quad b_{jy} + a_{ix}\sin\theta_0 + a_{iy}\cos\theta_0, \theta_0)$$

where $a_i = (a_{ix}, a_{iy})$ and $b_j = (b_{jx}, b_{jy})$ are vertices of A and B, respectively. The maximal x value must be achieved at one of these points.

The maximal x value for these points can occur, over the interval $[\theta_1, \theta_2]$, either when $\theta_0 = \theta_1$, $\theta_0 = \theta_2$, or when the derivative $(-a_{ix}\sin\theta_0 - a_{iy}\cos\theta_0)$ is zero. The latter occurs when

$$\sin(\theta_0 + \eta_i) = 0$$

which means

$$\theta_0 = n\pi - \eta_i$$

for some integer n. (Recall that η_i is the angle of the ray from the origin to a_i relative to the x axis in the local coordinate system of moving object A, whence $a_{ix} = \|a_i\|\cos\eta_i$ and $a_{iy} = \|a_i\|\sin\eta_i$.) Therefore, a least upper bound for x can be found by maximizing over b_{jx} for all j and maximizing over $a_{ix}\cos\theta_0 - a_{iy}\sin\theta_0$ for all i and θ_0, from among the three values given above. The bound for x is the sum of these two maxima. Similarly, a lower bound for x can be found.

The y case is completely analogous, except that the second expression to be maximized

is $a_{ix} \sin\theta_0 + a_{iy} \cos\theta_0$ and the derivative is zero valued when

$$\cos(\theta_0 + \eta_i) = 0$$

whence

$$\theta_0 = (n + 0.5)\pi - \eta_i$$

for some integer n.

3.5 Constructing an Initial Representation of Configuration Space.

The cells and associated constraint sentences constructed independently above for each pair of moving object and fixed obstacle polygons, must be embedded in a common configuration space. The technique used in the two-dimensional (fixed orientation) configuration space can be extended to the three-dimensional configuration space. The coordinates are scanned in some order and space is cut normal to the axis at every boundary of an embedded cell. The extent of the cuts is bounded by cuts already made for earlier coordinates in the sequence. There is an efficiency problem in using this straightforward generalization, however. None of x, y, or θ are attractive for the first direction of cutting.

Consider the θ direction. In general, differently shaped or oriented obstacles will give rise to different sets of cuts in the θ direction for their respective $CO_A^\theta(B)$. Since every $CO_A^\theta(B)$ spans the whole range of θ, they will all be cut by every θ cut of every other configuration obstacle. This generates a tremendous number of cells.

The x and y directions do not suffer from the problem that every cut is felt globally by all obstacles. But, the "stack" of θ-slices of $CO_A^\theta(B)$ gives rise to a large number of very close x or y cuts, if either is chosen to be the first direction of cutting. (Refer to Figure 10.) If another configuration obstacle lies in the path of these cuts it will be sliced into many pieces, none of which reflect anything inherent in its local structure.

This problem is solved by first boxing the stack of θ slices for a single configuration obstacle into a single cell that extends over the range $[-\pi, \pi]$ in the θ direction. Such

cells are embedded into the configuration space that is sliced in the x and y directions. (A simple heuristic can be used to choose which direction should be cut first: if the larger component of the vector from the initial point to goal point is in the x direction, then it is best to slice the space normal to the y–axis, as this has the possibility of providing large empty corridors, parallel to the x–axis, along which the object can be moved with almost no analysis.) Then, within each cell, the cutting procedure is used all over again, first cutting normal to the θ–axis then normal to each of x and y.

The efficiency of the algorithm can be further enhanced by arranging for the second stage of cutting to be carried out only if it is strictly needed. This is possible because the second stage is a purely local computation and does not effect the global structure of the configuration space representation. A bounding cell is labeled *mixed* and, instead of a standard constraint sentence, it has associated with it a list of the θ slices that must be embedded within it. It is straightforward to allow for the intersection of such bounding cells when they are embedded in the configuration space, and to compute the correct slice list for the intersected regions. If the *mixed* cell is never examined during the search procedure, then the second stage cutting operations need never be carried out. Otherwise the bounding cell is cut when it first becomes a candidate path cell as found by the A^* search algorithm.

3.6 Deciding Where to Cut a Cell.

The same scoring function as was given in section 2.6 is used for three–dimensional configuration space. Cuts are proposed for each constraint in the constraint sentence for a cut normal to each coordinate axis: x, y and θ. Cuts are proposed wherever they would make the constraint pass through a vertex of the newly cut cells. For each edge parallel to the coordinate being considered for cutting, the constraint is "solved" for that coordinate. A score is assigned to the cut, and overall the cut with least score is chosen.

4. Search

The representation described above for configuration space is useful only if some efficient way can be found to search it for collision-free paths. Our algorithm uses the A^* algorithm as its primary search engine to search the cell connectivity graph, both for paths through purely empty cells and for paths that include *mixed* cells.

Once a set of *empty* cells linking the initial point to the goal has been found, an actual point path through those cells must be chosen. Again the A^* search procedure is used, but this time using a selected set of points in the cells along the solution path.

If no path through *empty* cells is available, then the representation of configuration space must be refined through cell division. It is necessary, however, to decide where the space should be refined, and how much effort should be expended in subdivision, before a new search is initiated. The point path search mentioned above is used to direct the refinement of configuration space as well as to produce a final solution path for the problem.

The efficiency of the overall search can be greatly improved by using the *divide and conquer* paradigm. There are some complications in this application, however, as each subproblem is capable of changing the global data base by refining the representation of space within its area of search. This turns out to be a significant problem and some care must be taken to minimize its adverse effects. There is an additional problem with divide and conquer in this application. The division of a problem into subproblems cannot guarantee that if the original problem is solvable then so are all the subproblems. A form of resource limited computation is used to back out of problem subdivisions that do not look promising.

4.1 Search Procedure.

The findpath algorithm as presented in overview in section 1 makes use of two almost identical search procedures. One is to search for a connected set of *empty* cells that includes

the cell containing the initial point in configuration space and the cell containing the goal point. If that fails then the search is repeated but this time *mixed* cells are also considered.

The engine for both these searches is the A^* algorithm, presented in Nilsson [1971]. This is essentially a breadth first search, where a graph is searched to find a path from one node to another. The search proceeds by extending partial paths from the initial node. At each iteration, the lowest cost partial path is chosen and each of the neighbors of its latest node is appended to the partial path to create an extended partial path. The evaluation function used to rank partial paths is the sum of the cost in following the partial path and an estimate of the remaining cost to get to the goal. It is well known that as long as the estimate of remaining cost is a lower bound on the true remaining cost then this search procedure yields the minimum cost path.

While searching for a path of connected cells, a candidate point path through each partial cell path is "constructed". Each adjacent pair of cells intersect in a cell of one lower dimension. In the case of two dimensions without rotations, the interaction cell is a line segment and the centroid its midpoint, for example. The centroids of such interaction sub-cells are used as the entry and exit points of the path through the cell, and a the path segment within a cell is the straight line joining them.

The cost of an individual path segment is just its Euclidean length within the configuration space. Thus the cost assigned to a partially developed path is the sum of the costs of its path segments. The heuristic estimate of the cost of completing it, to reach the goal point, is just the Euclidean length from the last point on the path to the goal point. Clearly (by the triangle law) this is a lower bound on the actual cost. Furthermore, it has obvious heuristic merit.

The running time of the search algorithm can be improved by making the cost of traversing a *mixed* cell greater than the cost of traversing an *empty* cell. The relative cost can be used to trade off running time of the search algorithm for length of the final path. With a large relative cost, the A^* algorithm will strongly tend to concentrate on *empty* cells.

Hence, paths that require only a small amount of additional trail blazing through *mixed* cells will be chosen, even if they are very long. In effect, A^* will tend to skirt unexplored areas of the search space and stick with known empty space.

The above cost function treats all dimensions of the configuration space uniformly but, in cases where rotations are allowed, there is no *a priori* reason that this should be so. One can vary the weight given to rotational components (essentially stretching or compressing the configuration space parallel to the rotation axes) in computing the Euclidean length of the path segments within the space. This has no physical meaning in the original space, but can be used to control the amount of rotation in the final path. When low weight is used for rotations the final path tends to have seemingly arbitrary rotational perturbations back and forth. Higher weight for rotations tends to smooth out the paths.

4.2 Choosing a Point Path.

When a final cell path consisting of *empty* cells has been found it is still necessary to find a path for the configuration point of the moving object through configuration space. In general, the choice of final path should be based on domain–specific considerations such as kinematic or dynamic characteristics, not on purely geometric criteria. In the absence of domain–specific considerations we suggest that a desirable property of the final path is that it be smooth, essentially nulling out the artificial effects of the tesselation of configuration space by the cell structure. Our approach is to consider alternative paths going through a small set of candidate points in the interaction cell between adjacent cells in the chosen path, and to choose the one that minimizes the cost function.

Each pair of adjacent cells intersect in a cell of dimension one lower than that of the configuration space. A few points are chosen from that interaction sub–cell as candidate exit and entry points. For a two–dimensional configuration space the sub–cell is a line segment; reasonable choices are the two end points and the midpoint. For three–dimensional configuration space the sub–cell is a rectangle: in our implementation we used nine points, the vertices, midpoints of the edges, and centroid of the rectangle.

A graph can be constructed linking pairs of entry and exit points for each cell. The graph is searched using the A^* algorithm with the same cost function as detailed above. Within each cell the point path is a single straight line that connects the chosen points.

4.3 Refining the Search Space.

After a search has found a sequence of *empty* and *mixed* cells, the *mixed* cells along the path should be cut into at least two sub-cells and the search re-executed. This simple strategy leads to a very slowly converging search algorithm, however. The ratio of cutting to searching is too low. Arbitrarily cutting cells can also be wasted effort, however, so it is better to find some selective method of determining places to make more cuts. We use the point path search to identify such places.

The point path search of the previous section is used to find a point path through the *empty* and *mixed* cells. All points at the interface between cells on that path are then examined. It is easy to determine if a point is actually in free space by evaluating the constraint sentence associated with one of the containing cells. If the point is actually in free space, but its cell is mixed, then the cell is repeatedly divided until the point lies in a newly created *empty* cell. (This same procedure is used to cut the space representation about the initial and goal points to get *empty* initial and goal cells for the original search.) The effect of this is to create small islands of *empty* cells along the trajectory, which will be useful for defining recursive subproblems (see next section).

Initially there may be many large cells in the configuration space representation. If an arbitrary point on the interface between two adjacent cells were chosen as a center of refinement of the cell representation, it could well be that it is far from the optimal path. By only expanding points on a candidate optimal path, the overall search operation is not misled into grossly suboptimal areas of "easy pickings".

4.4 Divide and Conquer.

The efficiency of the search for a *free* cell path can be greatly improved by hierarchically decomposing it into more localized and smaller problems. The search space is smaller for a localized problem, resulting in reduced time for the overall search.

The problem is how to break up a global search into smaller ones when there is no a *priori* knowledge of what the smaller subproblems are. The approach taken here is to use the global search through *mixed* cells, and subsequent selection of a point path as a plan for the lower level subproblems. If there are points on the path that, after cell refinement as above, are in a known *empty* cell, then they are used as points at which the problem is broken up. The global problem then becomes a series of local ones, each from an initial point to a goal point where both are in *empty* cells.

Surprisingly, the order in which the subproblems are attempted is important. This is because each subproblem may require that the global data base, the cell connectivity graph, be altered, as *mixed* cells are cut. If the subproblems are evaluated with the one closest to the global initial point first, and followed sequentially by those closer and closer to the goal point, then this cutting has adverse effects on search efficiency.

Consider the first subproblem in a sequence. In general, many cells near the goal point will have been refined by the time a sub–path through *empty* cells has been found. Now the second subproblem is attacked. Its initial point is the goal point of the previous subproblem. When the A^* search starts at that point it finds many small *empty* cells from the recent search. These cells are in the wrong direction relative to the new goal, however; they are, nevertheless, "attractive" to the search algorithm, because the evaluation function favors *empty* cells. In addition the *empty* cells are small and, therefore, many can be appended to partial paths near initial point without sharp increases in the heuristic estimate of remaining cost. The result is that the A^* algorithm can spend inordinate amounts of time breadth–first searching backwards from the new initial point over areas covered by the previous subproblem.

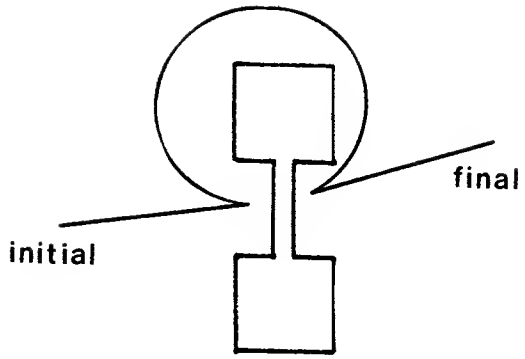


Figure 11. If no restriction is placed on the path length used to solve a subproblem, very inefficient paths may be found as illustrated here.

Fortunately, this wasted work can be avoided by solving the subproblems in the reverse order – taking the one nearest to the global goal point first. This change avoids the problem because the many small empty cells produced by previous problems are only encountered at the end of the search. By then, the heuristic estimate of remaining cost tends to dominate the differences in the evaluation function. The *empty* cells can only be reached by going past the goal, so other cells along the way have smaller costs and will be developed first.

Another consideration in the use of divide and conquer is the possibility that a subproblem may be poorly chosen so that there is no very direct path between the chosen initial and goal points. This can be the case even when the global problem is soluble. If unlimited effort is expended on solving the subproblem then it can happen that the final path "backs" out from the initial point, finds a path around the obstacles that block a more direct route from the initial to final position, then "backs" into the goal position. The two subproblems on either side backtrack over those initial and final parts of the path. See Figure 11 for an illustration of this effect.

In such cases, after some effort has been expended to determine that the direct path is blocked, it would be better to back up to the more global problem and look for a new path through mixed cells. This can be simply achieved by placing a path length limitation

on subproblems. As the search space is refined and new optimal paths are found through *mixed* cells their cost is compared with the limit. Once it is exceeded the subproblem exits with failure, forcing a new search at a higher level. In the examples given in this paper the cost limitation for subproblems is 1.5 times the cost of the global path, or 1.5 times the best cost ever computed for the sub–path, whichever is smaller.

5. Conclusion

The algorithm described here has been implemented and tested on many randomly generated examples. Figure 12 shows two difficult cases that the algorithm solves. We believe these to be among the most difficult findpath problems ever solved by a program. The running time of the algorithm on these examples is very high, however. The actual running times for these difficult problems are on the order of tens of minutes of "wall–clock" time on a single–user MIT Lisp Machine without floating point hardware; these times also include a very significant paging overhead. Simple problems, of course, run much faster: on the order of tens of seconds to a few minutes. This should be contrasted with running times on the order of less than a minute for problems of moderate complexity using the algorithm described in Brooks [1982]. This latter algorithm cannot solve problems such as illustrated in Figure 12, however.

While the running times of the algorithm described here could be made significantly shorter by implementation changes, the fact remains that the complexity of the algorithm is high. Informal timing experiments seem to indicate that the algorithm spends much of the running time in the A^* search procedure, which is used at several points in the algorithm. If the number of cells to be searched could be reduced, the running time could be significantly reduced. One approach to doing this is to use an algorithm such as described by Brooks [1982] to identify a likely path, using a smaller and simpler moving object, and then apply the algorithm described here to verify and refine the path for the actual moving object. This would reduce the size of the search space for the algorithm. There are a number of conceptual and technical problems to be solved before this hybrid approach is practical. As of now, the algorithm described here can solve very complex problems, albeit slowly.

The approach followed in this algorithm can be directly applied to configuration spaces for three-dimensional polyhedra whose orientation is fixed. This case generates a three–dimensional configuration space with linear constraints, a direct generalization of Section 2. We believe that the generalization to a four–dimensional configuration space, such as for a polyhedra with a single rotational degree of freedom, will be straightforward. A new

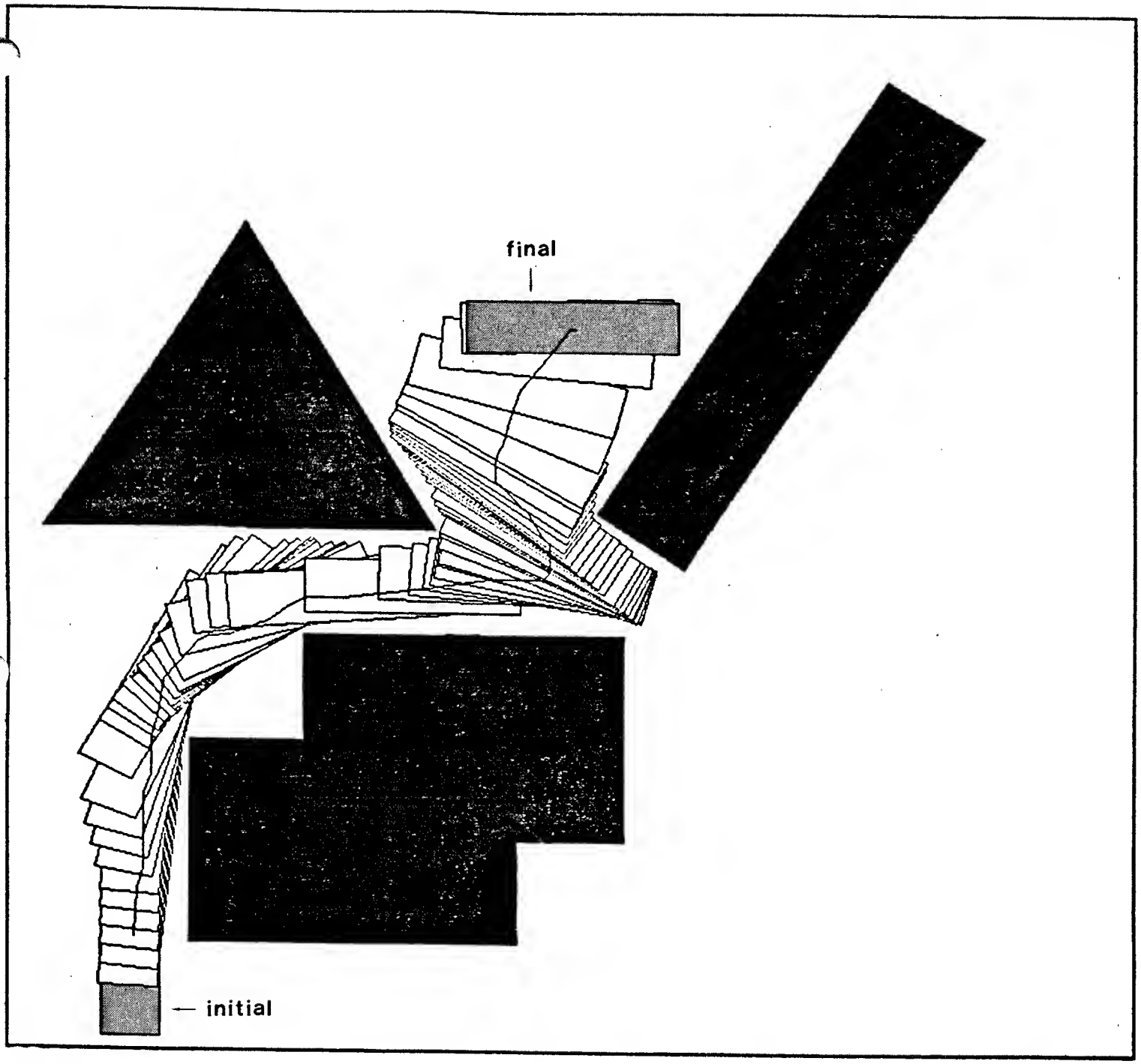


Figure 12a. A path found by the algorithm for a convex moving object. The final representation of configuration space for the problem has 766 cells of which 122 are full, 524 are mixed and 120 are empty. The cells are linked by 2157 connecting arcs and the final path goes through 87 of the empty cells (i.e. 71.3%).

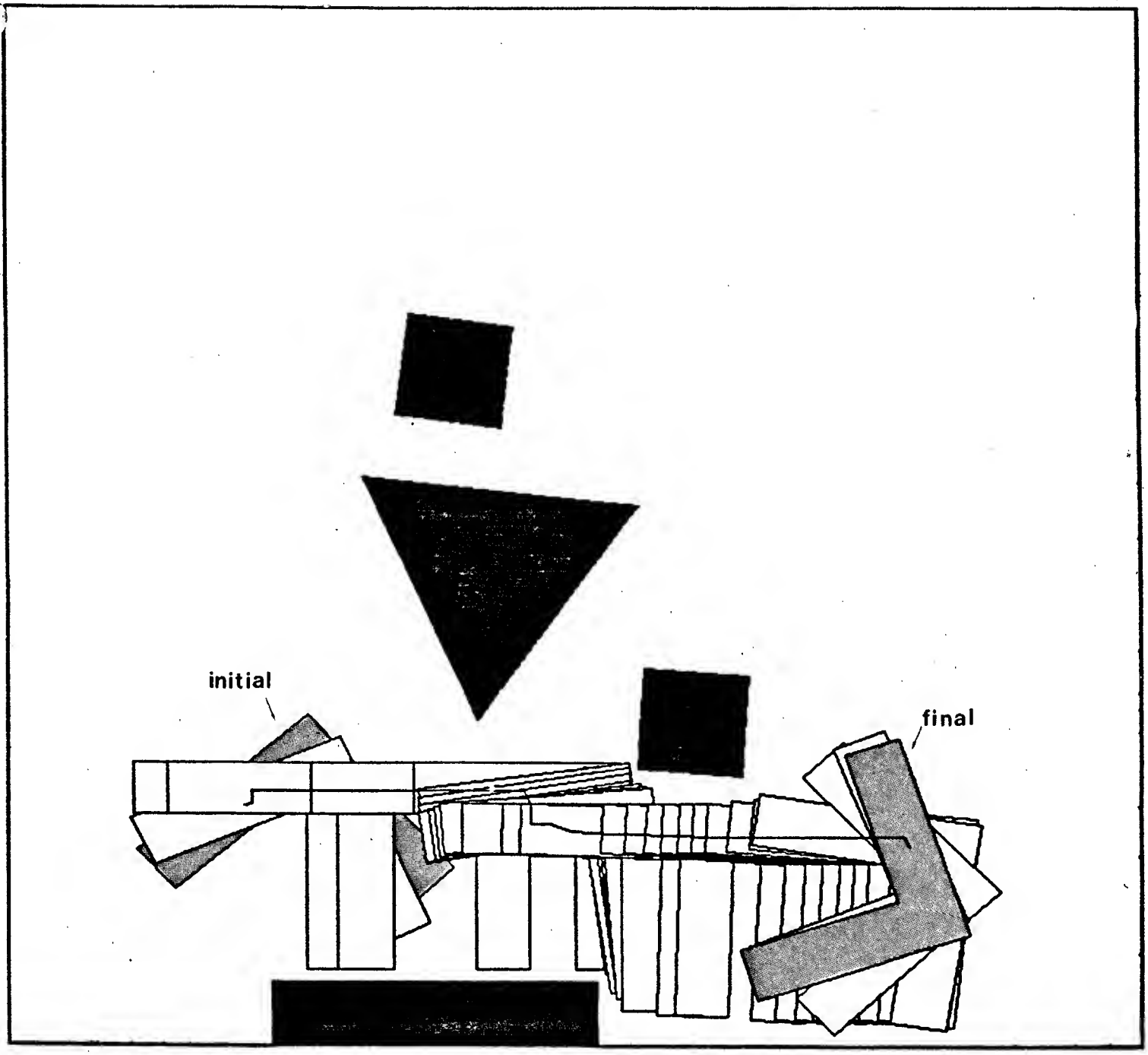


Figure 12b. A path found by the algorithm for a non-convex moving object. The final representation of configuration space for the problem has 892 cells of which 110 are full, 720 are mixed and 62 are empty. The cells are linked by 2489 connecting arcs and the final path goes through 29 of the empty cells (i.e. 46.7%).

type of constraint surface must be dealt with, however, arising from the interaction of pairs of edges (Lozano-Pérez [1983]).

The approach could, in principle, also be generalized to configuration spaces of higher dimension, such as those for polyhedra that are allowed to rotate. The actual generalization presents a large number of problems, e.g., the CellCompare operation is substantially more difficult, and the number of cells grows extremely fast. Other algorithms suggested for this general case have the same drawback. The algorithm for the general findpath problem given by Schwartz and Sharir [1982], for example, has a very high polynomial time complexity for a fixed number of degrees of freedom and is exponential in the degrees of freedom. Our belief is that, in practice, the general six degree of freedom problem should be heuristically reduced to cases involving four or fewer degrees of freedom.

References

Brooks, Rodney A. (1982). *Solving the find-path problem by representing free space as generalized cones,* Artificial Intelligence Laboratory, Massachusetts Institute of Technology, AI Memo 674, May.

Lozano–Pérez, Tomás (1981). *Automatic Planning of Manipulator Transfer Movements,* IEEE Trans. on Systems. Man and Cybernetics (SMC–11):681–698.

——— (1983). *Spatial Planning: A Configuration Space Approach,* IEEE Trans. on Computers, *in press.*

——— (1983). *Task Planning,* in *Robot Motion: Planning and Control,* M. Brady, *et al.* eds., MIT Press.

Lozano–Pérez, Tomás and Michael A. Wesley (1979). *An algorithm for planning collision-free paths among polyhedral obstacles,* Communications of the ACM (22):560–570.

Moravec, Hans P. (1980). *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover,* Ph.D. Dissertation Stanford AIM-340, Sept.

Nilsson, Nils J. (1971). *Problem–Solving Methods in Artificial Intelligence,* McGraw-Hill, 1971.

Schwartz, Jacob T. and Micha Sharir (1981). *On the Piano Movers Problem I: The Case of a Two-Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers,* Department of Computer Science, Courant Institute of Mathematical Sciences, NYU, Report 39, October.

Schwartz, Jacob T. and Micha Sharir (1982). *On the Piano Movers Problem II: General Properties for Computing Topological Properties of Real Algebraic Manifolds,*

Department of Computer Science, Courant Institute of Mathematical Sciences, NYU, Report 41, February.

Udupa, Shriram M. (1977). *Collision Detection and Avoidance in Computer Controlled Manipulators,* Proceedings of IJCAI-5, MIT, Cambridge, Ma., Aug. 1977, 737-748.